SFA Modernization Partner

United States Department of Education

Student Financial Assistance

ITA
Best Practices Guide

Task Order #46

Deliverable # 46.1.5

Version 1.0

September 28, 2001

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to collect and document the Best Practices that have been identified and used within the Java Based WebSphere Application Server version 3.5.3 environment at SFA . These practices are a collection of recommendations that have been passed down from IBM and Accenture in their in-depth experience with E-Commerce and its implementation within Java based Application Servers.

1.2 Intended audience

The ITA Best Practice document is aimed at the audience of technical architects, designers, and application developers who are responsible for creating Java-Based solutions within the SFA WebSphere environment. The document assumes that the reader is knowledgeable in Java and WebSphere Application Server and has access to the Java 2 Software Development Kit (JSDK) and WebSphere reference documents.

1.3 Background

In assisting with production releases of previous SFA web applications, the Integrated Technical Architecture team determined that it was necessary to collect best practice white papers on development and administration of applications that run within the SFA WebSphere environment. This allows SFA to benefit from previous work done at other WebSphere Customer sites that have done benchmarking and performance testing work to identify the best practices. These best practices help SFA design their applications to best use WebSphere and to ensure quality and scalable Web sites.

1.4 Scope

These guidelines will provide specific examples and recommendations for integrating key application architectural functions including database connection pooling, session management, object persistence, and profile management.

1.5 Assumptions

The current WebSphere environment at SFA includes WebSphere 3.5.3, its associated Web Server (IBM HTTP Server 1.3.12), Oracle databases(8.1.7), and Java Development Kit (1.2.2). These recommendations and best practices may and will change with later releases of any these products.

2 Session Management

HTTP is a stateless protocol because each command is executed independently, without any knowledge of the commands that came before it. Since HTTP treats each user request as a discrete, independent entity, HTTP alone does not recognize or maintain a user's state. As a result, a Web application needs a mechanism to hold the user's state information over a period of time (typically known as a visit).

The Java Servlet specification (http://java.sun.com/products/servlet/) provides a mechanism for servlet applications to maintain a user's state information. This mechanism, known as a session, allows a Web application to maintain all user state information at the server, while passing minimal information back to the client (web browser) via cookies or another technique known as URLencoding.

By default, WebSphere places session objects in memory. However, the administrator has the option of enabling persistent session management, which instructs WebSphere to place session objects in a database as well.

Administrators enable persistent session management when:

- Multiple WebSphere instances need to share session objects (also known as clustering).
- The user's session data becomes too valuable to lose through unexpected failure at the WebSphere node.
- The administrator desires better control of the session cache memory footprint. By sending cache overflow to a persistent session database, the administrator controls the number of sessions allowed in memory at any given time.

Here are some guidelines for session management:

2.1 Do not combine the Session Management DataBase with other databases.

When configuring persistent sessions in WebSphere Application Server in production, use a dedicated data source. To avoid contention for JDBC connections, do not reuse an Application DataSource or the WebSphere Application Server repository for persistent Session data.

2.2 Be Aware of Your Servlets Accessing A Session Concurrently.

WebSphere version 3.5.3 and above complies with the servlet 2.2 specification that removes the requirement for WebSphere to lock the session upon intent to update. The developer now has to be aware of servlets within a particular session accessing the same session concurrently**.** Issues that usually occur with this situation are two servlets changing the same session property at the same time or one servlet invalidating the session while another one is still trying to use it. Solutions for this problem can involve the following:

- Good Architectural planning for the Application's Session Data Management
- Naming standards for session variables that ensure uniqueness among servlets or Web Applications.
- Developers may consider synchronizing on the session following the request.getSession call, with possibly checking the validity of the session as the first step following that Java synchronization. This will make the application less multithreaded but should ensure session validity.

2.3 Serializable Objects

All information stored in a persistent session database must be serializable. All of the objects held by a session must implement java.io.Serializable. In general, it is an excellent idea to always serialize any new session objects even if persistence sessions are not being used. This would allow a site that is not using persistence sessions to start using it immediately without code changes. An example of how to serialize a session object is below.

```
public class MyObject implements java.io.Serializable
```

2.4 Do Not Store Large Objects in HttpSession

A persistent HttpSession must be read by the servlet whenever it is used and rewritten whenever it is updated. This involves serializing the data and reading it from and writing it to a database. Performance of the WebSphere Session management drops dramatically, as the session object becomes larger and larger. It is critical that an application minimize the size of the session object as much as possible. The ideal case is that the session object is less then four kilobytes in size.

2.5 Release HttpSessions When Finished

HttpSession objects live inside the WebSphere servlet engine until:

- The application explicitly and programmatically releases it using the API, **javax.servlet.http.HttpSession.invalidate (**);
- WebSphere Application Server destroys the allocated HttpSession when it expires after non-use for 1800 seconds (by default).

Due to the fact that Websphere's Session cache can only maintaine a defined number of session objects, it is critical that applications take advantage of WebSphere's Session management to release sessions when the sessions are no longer needed. A session can be released using the invalidate() function specified above. Developers must ensure the application is completely finished with the session before it is released.

2.6 Do Not Create HttpSessions in Java ServPages (JSPs) By Default

By default, Java ServerPages (JSP) files create HttpSessions. This is in compliance with J2EE standards to facilitate the use of JSP implicit objects, which can be referenced in JSP source and tags without explicit declaration. HttpSession is an example of implicit objects. If the developers do not use HttpSession in the JSP files, then the developers can save some performance overhead with the following JSP page directive:

```
<%@ page session="false"%>
```

2.7 Use Session Affinity

Proper use of session affinity can enhance the performance of WebSphere. Session affinity in WebSphere is a way to maximize the use of the in-memory cache of session objects and reduce the amount of reads to the backend database. Session affinity works by caching the session objects in the JVM of the application a user is interacting with. If there are multiple clones of this application, the user can be directed to any one of the clones. If the users starts on clone1 and then comes in on clone2 a little later, all the session information must be persisted to the backend database and then read in by the JVM that clone2 is running in. This database read can be avoided by using session affinity. With session affinity, the user would start on clone1 for the first request and then for every successive request, the user would be directed back to clone1. By doing this, clone1 only has to look at the cache to get the session information and never has to make a call to the session database to get the information.

Designers and developers can improve performance by not breaking session affinity. Here are a few suggestions to help prevent breaking session affinity:

- Do not use multiframed JSPs where the frames point to different web applications. This will break affinity and will cause separate JVMs to process a session concurrently. When this happens, consistent state cannot be guaranteed.
- If possible, combine all web applications into a single JVM (Application Server) and use modeling / cloning to provide fail-over support.

2.8 Use of Manual Update and Sync()

When an application is using HTTPSession, anytime data is read from or written to that session, the Last Access time field is updated. If persistent sessions are enabled, this produces a new write to the database. If the application is mostly an HTTPSession Read-only application then this performance hit that can be avoided by using Manual Update and having the record written back to the database only when data values are updated, not on every read / write of the record.

To use manual update, Websphere administrator needs to turn it on in the session manager (Available through the admin browser). Additionally, the application code must use the *com.ibm.websphere.servlet.session.IBMSession* instead of the generic *HttpSession*. Within the IBMSession there is a method called sync(). This method tells WebSphere that the data in the session object should be written out to the database. This allows the developer to improve overall performance by having the session information persist only when necessary.

3 Database Access

Each time a web resource attempts to access a database, it must connect to that database. A database connection incurs overhead -- it requires resources to create the connection, maintain it, and then release it when it is no longer required.

The overhead is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter, due to the surfing nature of the Internet. Often, more effort is spent connecting and disconnecting than is spent during the interactions themselves. Further, because Internet requests can arrive from virtually anywhere, usage volumes can be large and difficult to predict.

Here are some guidelines for database access:

3.1 Use JDBC Connection Pooling

To avoid the overhead of acquiring and closing JDBC connections, WebSphere Application Server 3.5.3 provides JDBC connection pooling based on JDBC 2.0. Benefits of using connection pooling are:

- Allows the administrator control and reduce the resources used by Web-based applications.
- Spreads the connection overhead across several user requests, conserving resources.
- Improves the response times of Web-based applications.

Applications should use WebSphere Application Server JDBC connection pooling instead of acquiring these connections directly from the JDBC driver.

3.2 Perform expensive JNDI lookups once per data source

Even though using connection pooling does improve database connection performance dramatically, performance can be improved even more by reducing the amount of times a Java Naming and Directory Interface (JNDI) lookup is done. JNDI lookups are expensive from a performance perspective so a developer should minimize the amount of times JNDI lookups are done. To minimize JNDI lookups, JNDI calls can be performed in an ***init*** method of a servlet or doing the lookup within an initialization method of a class. A common procedure is to obtain and store the datasource as a private class variable. If the private variable is set to null initially and then checked each time before it is used, the developer can perform the lookup only if it is still null.

In JDBC 2.0, a Naming Service is used to locate a data source. An initial naming context is obtained, and then the data source is located by doing a JNDI lookup. The data source "name" provided on the ctx.lookup(name) is not the physical name of the data source, but is the logical name for the data source that has been defined administratively. This allows change to the JDBC driver to be updated administratively rather than by code compilation. A sample of how to access a database connection via jdbc pooling is below.

```
try {
        java.util.Hashtable env= new java.util.Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejs.ns.jndi.CNInitialContextFactory");

        // Create the Initial Naming Context.
        ctx = new InitialContext(env);

        // Perform a naming service lookup to get a DataSource object.
        ds = (javax.sql.DataSource) ctx.lookup(dbJndi);
}
catch (Throwable theException) {
        // Handle Exception
}
```

3.3 Release JDBC Resources When Done

When JDBC resources are no longer needed, these resources should be released and/or closed -- ensure that they are closed/released in all circumstances, including exceptions and errors. JDBC connection pooling provides a defined and finite number of JDBC connections. Below is a list of things to note when working with JDBC resources:

- JDBC connection shortages cause long waits.
- Failing to close JDBC connections causes waits for idle JDBC connections to be reaped and reused.
- Closing JDBC connections will allow quicker reuse and improve performance.
- Failing to close and release JDBC connections can cause other users to experience long waits for connections.
- Although a JDBC connection that is left unclosed will be reaped and returned by WebSphere Application Server after a timeout period, others may have to wait for this to occur.
- Close JDBC statements when the application is through with them.
- JDBC ResultSets can be explicitly closed as well. If not explicitly closed, ResultSets are released when their associated statements are closed.

- Ensure that the code is structured to close and release JDBC resources in all cases, even in exception and error conditions.

The proper way to close JDBC Connections and PreparedStatements:

```
Connection conn = null;
ResultSet rs = null;
PreparedStatement pss = null;
try
{
        conn = dataSource.getConnection(USERID,PASSWORD);
        pss = conn.prepareStatement("SELECT SAVESERIALZEDDATA
            FROM SESSION.PINGSESSION3DATA WHERE SESSIONKEY = ?");
        pss.setString(1,sessionKey);
        rs = pss.executeQuery();
        pss.close();
        conn.close();
}
catch (Throwable t)
{
        // Insert Appropriate Error Handling Here
}
finally
{
        // The finally clause is always executed - even in error
        // conditions PreparedStatements and Connections will always be closed
        try
        {
                if (pss != null)
                        pss.close();
        }
        catch(Exception e) {}

        try
        {
                if (conn != null)
                        conn.close();
        }
        catch (Exception e){}
        }
}
```

3.4 Specify database attributes at deployment time

The database attributes such as the user ID, password, and database source name should be modifiable administratively without requiring code changes. This simplifies the process of migrating from a staging environment to a production environment. This can be accomplished by including these parameters as InitParameters for a servlet or by reading these parameters from a properties file or resource bundle.

3.5 Make Use of PreparedStatement When Possible

Each time the application submits a query statement to a database, the database manager creates a query plan. The query is then executed with this plan. If the application is coded to use the PreparedStatement class instead of the Statement class, an object is created that precompiles and stores the SQL (Structured Query Language) associated with the query plan. This approach is more efficient than running the same statement multiple times with a Statement object, which compiles the statement each time it runs. WebSphere maintains a cache of precompiled PreparedStatement objects. The default cache size for PreparedStatement objects is 100. If the application uses more than 100 different PreparedStatements, the cache size can be modified using the datasource.xml file.

3.6 Application Recovery from Database Failures

A connection pool is a cache of database connections. This cache could be rendered stale/invalid in the following cases:

- Scheduled database shutdowns
- Database administrative actions such as issuing the command force applications all with DB2 or shutdown with Oracle
- Network problems

A WebSphere supported database driver will execute the following on all SQL Exceptions.

1. Catch SQLException on database operations
2. Examine the exception for SQLSTATE string, error codes
3. Compare with list of known state strings and error codes
4. Destroy the connection

To allow seamless processing a retry loop should be included in the application program. The developer will need to write a catch block which can catch an SQLException called “StaleConnectionException” and then retry the connection. An example is below:

```
try {
   boolean retry = true;
   while( retry) {
      try {
         Connection conn1 = ds1.getConnection();
         Statement stmt1 = conn1.createStatement();
         // assume that the connection becomes stale here somehow.
```

```
                    // eg. force application all.
                    // the following call will fail.
                    ResultSet res = stmt1.executeQuery(stmtString);
                    .......
                    // if all the database work was successful.
                    retry = false;
                    ............
                    }
                    catch (StaleConnectionException ex) {
                       conn1.close();
                       retry = true;
                    }
           }
        }
        catch( Exception ex) {
           // handle exceptions
        }
```

If WebSphere is using a generic JDBC driver to support the JDBC connection to a database, then the instead of catching StaleConnectionException, the developer will need to catch all SqlExceptions and decide if retry is possible.

4 Performance

Application performance can be improved if the following guidelines are implemented:

4.1 Use the HttpServlet Init method to perform expensive operations that need only be done once

Because the servlet init() method is invoked when servlet instance is loaded, it is the appropriate location to carry out expensive operations that need only be performed during initialization. By definition, the init() method is thread-safe. The results of operations in the HttpServlet.init() method can be cached safely in servlet instance variables, which become read-only in the servlet service method.

4.2 Use calls to ServletContext.log() sparingly

Each call to the ServletContext.log() method is recorded in the WebSphere administrative database. Overusing calls to this method will seriously degrade performance. Limit calls to only those events that should be considered seriousEvents.

4.3 Use Session Cache

By enabling WebSphere Session Cache, Websphere will maintain a list of the most recently used sessions in memory and avoid using the database to read in or access the session when it determines that the cache entry is still the most recently updated copy. This will give WebSphere a big performance kick compared to using persistent sessions only. It is very important to accurately estimate the amount of sessions that WebSphere will handle concurrently so that the administrator may set up the maximum Java Virtual Machine memory size as well as the WebSphere Session cache size accurately. WebSphere allows the administrator to define a limit on the number of sessions held in the in-memory cache via the administrative console settings on the Session Manager. This prevents the sessions from acquiring too much memory in the Java Virtual Machine associated with the WebSphere instance. The default is 1000 concurrent sessions but administrators must use current web reports to estimate maximum concurrency.

The WebSphere Administrator does have the capability to build a second in-memory cache in the case that users overrun the primary cache mentioned above. This second cache can be enabled by selecting *Allow Overflow* in the Session Manager on the Admin console. Although this second cache will use all available memory to fulfill session requests beyond the value specified in the Base memory limit, it does have the possibility to bring down the Web Server since no limits can be placed on it. Also when a Session database is in use, the overflow cache is sent to the database instead of memory cache. Due to these concerns, the use of the overflow cache should be considered with great care.

4.4 Minimize use of System.out.println

Because it seems harmless, this commonly used application development legacy is overlooked for the performance problem it really is. System.out.println statements and similar constructs synchronize processing for the duration of disk I/O, so they can significantly slow throughput.

The RCS Logging framework allows logging to be written to memory buffers and lazy written to log files. This lessens the reliance on synchronized I/O writes to the disk thus improving the performance of the application.

4.5 Avoid String concatenation "+="

String concatenation is the textbook bad practice that illustrates the adverse performance impact of creating large numbers of temporary Java objects. Because Strings are immutable objects, String concatenation results in temporary object creation that increases Java garbage collection and consequently CPU utilization as well. The textbook solution is to use java.lang.StringBuffer instead of string concatenation.

```
res.setContentType(“text/HTML”);
PrintWriter out = res.getWriter();
String aStudent = “James Bond”;
String aGrade = “A”;
StringBuffer strBuf = new StringBuffer();
strBuf.append(aStudent);
strBuf.append(" received a grade of ");
strBuf.append(aGrade);
System.out.println (strBuf);
```

4.6 Do not use SingleThreadModel

SingleThreadModel is a tag interface that a servlet can implement to transfer its re-entrancy problem to the servlet engine. As such, javax.servlet.SingleThreadModel is part of the J2EE specification. The WebSphere servlet engine handles the servlet’s reentrancy problem by creating separate servlet instances for each user. Because this causes a great amount of system overhead, SingleThreadModel should be avoided.

Developers typically use javax.servlet.SingleThreadModel to protect updateable servlet instances in a multithreaded environment. The better approach is to avoid using servlet instance variables that are updated from the servlet’s service method.

4.7 Minimize Garbage Collection Activities

If the application constantly allocates objects up to the size of the heap, the garbage collector will be called more frequently. Reducing the number of times the garbage collector is called can improve the runtime performance of the application. For example, consider a method that creates temporary objects inside a loop when it could create a single object outside the loop and reuse it within the loop. By changing the method to reuse an object, the overhead of the garbage collector running can be avoided.

4.8 Use Buffered I/O

Using unbuffered I/O causes a lot of system calls for methods like InputStream.read(). These methods synchronize processing for the duration of disk I/O, and can significantly slow throughput.

4.9 Try to Avoid New

JVM internal synchronization caused by the new operation can cause lock contention for applications with lots of threads. Sometimes the new operation can be avoided by re-using byte arrays, or re-using objects that have some notion of a state-resettng method.

5 Object Persistence

Object persistence functionality can be made easier for application design and development by following the suggestions below.

5.1 Use RCS Persistence Framework

The RCS Persistence framework provides services that allow applications to interact with database(s) via the application server in order to create, retrieve, update, and delete business objects. The RCS Persistence framework implements many of the best practices suggested in Section 3 Database Access of this document and should be used for object persistence activities. Please refer to the Persistence user guide for more details.

5.2 Use Optimistic Record Locking

System transactions must have a very short duration to avoid unacceptable resource contention (i.e., system transactions and database locks should not span user think time). The use of naive pessimistic locking strategies to ensure the integrity of business transactions results in significant locking overhead and impaired system throughput and performance. Unfortunately, pessimistic locking underlies the concurrency control mechanisms commonly provided by most database systems.

For example, a user needs to update the customer information and goes through these steps:

- Get a copy of customer data (first transaction)
- Modify this copy
- Send the copy to the server side to make this update permanent (second transaction)

It could happen that another user concurrently accesses the same data for update as described in this scenario:

1. userA requests a copy of customerA’s data
2. userB requests a copy of customerA’s data
3. userA changes customerA’s name from ‘Kurt Weiss’ to ‘Martin Weiss’
4. userB changes customerA’s birthdate from 57/05/01 to 55/08/08
5. userA sends the changed copy for server-side update
6. userB sends the changed copy for server-side update

How can the application prevent the updates made by userB from overriding those of userA?

The common solution to this problem for traditional applications is through the use of a time-stamp (or version number) field on each table representing an entity subject to concurrency conflicts. The time-stamp field is updated whenever a record is successfully inserted or updated (e.g., through the use of triggers). Applications are responsible for checking that the time-stamp field has not changed between the time when the record was retrieved and the time the update is attempted. A common way to perform this check is to use a " SELECT FOR UPDATE " clause in SQL update statements, which ensures the update only goes through if the record's time stamp in the database matches the value supplied in the " SELECT FOR UPDATE " clause.

The Persistence framework supports optimistic record locking capability.

6 Profile Management

Personalization describes a range of features that enable applications to treat users / visitors as particular individuals. For a really simple example, consider a site that issues the message "Hello, John Smith" when the customer John Smith logs onto the site.

An example of Personalization that IBM WebSphere Application Server Version 3.5 provides is a service for processing user profiles -- the *User Profile Manage*r. The service is provided in the form of an EJB entity bean that servlets can call whenever they are required to access a user profile.

The key activities for implementing user profiles are summarized:

1. Customize the User Profile support as necessary. Options include:

   - Using the data representation class with exactly the name / value pairs it currently allows (no action required)
   - Extending the data representation class to allow additional, arbitrary name / value pairs
   - Adding columns to the base user profile representation
   - Extending the User Profile enterprise bean itself to import existing databases
   - Basically, you need to evaluate whether the user profile representation provided by IBM represents the kind of data you would like to keep about your users. You might find it desirable to customize the IBM user profile support in one or more of the above ways.

2. Create or modify servlets to use the User Profile Manager and related user profile support classes to maintain user profiles on behalf of Web applications.

3. Ensure the administrator appropriately configures User Profile Managers in the administrative domain.

4. If the developers and administrators are not the same people, the developers might need to provide settings information to the administrator, based on how the developers implemented user profiles.

7 Exception Handling Rules

Here are rules for exception handling within a Java application. These rules should be used in conjunction with the RCS Exception Handling framework.

7.1 Only Catch Exceptions That You Can Handle

In general, there are several ways of dealing with an exception when it occurs inside a method:

1. If the method cannot do anything about the exception, the exception is declared in the method's throws clause. This delegates the exception handling responsibilities to the caller of the method.
2. If the method can perform a partial recovery, it can do one of the following two things:
    1. Perform partial recovery and then re-throw the exception
    2. Perform partial recovery and throw a wrapped exception that contains the original exception
3. If the method can perform a full recovery, it can do one of the following two things:
    1. Perform full recovery operations directly inside the method (inline)
    2. Call a standard exception handler function to perform the recovery operations

When a checked exception occurs in the application code, it should either perform the full recovery operations directly within the method where the exception occurred or delegate the error handling responsibilities to the appropriate exception handling function.

7.2 Only Throw Custom Exceptions

Application code should always throw customized exceptions. Application code should never throw Java API exceptions, such as, IOException, ClassNotFoundException, and etc. These exceptions should be caught and handled appropriately.

7.3 Don't Re-Throw An Exception In A Way That Leads To A Loss Of Information

The following code example shows how a loss of information can occur in exception handling code:

```
try {
    Class cls = Class.forName( className );
}
catch ( ClassNotFoundException cnfe ) {
    // this throw leads to a loss of information.  Don't' do it.
    Throw new MyException( "An error has occurred" );
}
```

Because MyException is thrown out of the catch block, the original ClassNotFoundException is discarded. As a result, it will be very difficult to determine the real reason that the code in the try block failed. This kind of a loss of information should be avoided.

Either the original exception should be re-thrown (in the example above, simply throw cnfe; rather than creating a new exception of a different type), or it should be enclosed in the new exception being thrown.

The following example demonstrates how the Configuration framework does this correctly:

```
try {
    Class cls = Class.forName( className );
}
catch (ClassNotFoundException cnfe) {
    String msg = "An error has occurred";
    Throw new ConfigurationException( msg, cnfe );
}
```

7.4 Throw Checked Exceptions Over Unchecked Exceptions

In nearly all circumstances, applications should throw checked exceptions rather than unchecked exceptions because unchecked exceptions bypass the exception handling contract enforced by the compiler. Specifically,

- A method may throw an unchecked exception without including it in the throws clause.
- Callers are not required to handle unchecked exceptions, even if declared in the throws clause.

The rationale for unchecked exceptions for internal VM errors, linkage errors, and runtime violations of language semantics or security policies is sound, but it does not extend to application exceptions.

Although unchecked exceptions seem to offer a way to avoid cluttering the code with try-catch statements and throws clauses, the temptation is overwhelming but false for application code. The simple truth is that exceptions will occur. Failure to handle those exceptions properly makes an application unpredictable and unreliable.

One can document unchecked exceptions, but programmers can and will overlook them. With a checked exception, a programmer must actively ignore it by creating an empty catch block, and that is much easier to spot than an ignored unchecked exception. Bypassing the compiler support of the exception contract by using unchecked exceptions removes one important tool for enforcing proper exception discipline.

7.5 Application Code Should Never Directly Catch or Throw Unchecked Exceptions

Application code should not deal with unchecked exception directly. Unchecked exceptions will be handled by the last resort handler functions or by Webphere application server.

7.6 Order Catch Clauses With Sub-Classes Before Base Classes

```
// correct way to order catch clauses
try {
   FileReader input = new FileReader(filename);
   // use input…
}
catch (FileNotFoundException fnfe) {
   // perform specific handling when file not found…
}
catch (IOException ioe) {
   // perform more general handling for other IO exceptions…
}
catch (Exception e) {
   // very general handling for all other exceptions….
   // don't do this in the application code.  Developers should
   // always catch specific exceptions.
}
```

Java will transfer control to the first block whose catch clause matches the run-time exception type. This requires that a subclass be listed before its base class when both appear in the same try-catch block. Otherwise, the handler for the subclass will never be executed.

Exception type matches are based upon whether the exception thrown can be assigned to the parameter of catch clause. In the example above, FileNotFoundException can be assigned to fnfe, ioe, and e, but fnfe is the first match. EOFException, however, can only be assigned to ioe and e.

7.7 Do Not "eat" Exceptions

Error handling in Java is based on throwing and catching of exceptions. When an exception is thrown by a class method, it makes the statement: "method could not continue." This also implies that the method's post-conditions may not be met. This may leave the object in an undefined or unusable state, therefore application code should never catch an exception and then do nothing about it.

The following code example should NEVER be done in an application code:

```
// correct way to order catch clauses
try {
   FileReader input = new FileReader(filename);
   // use input…
}
catch (FileNotFoundException fnfe) {
   // do nothing
}
```

8 Model View Controller (MVC) Pattern

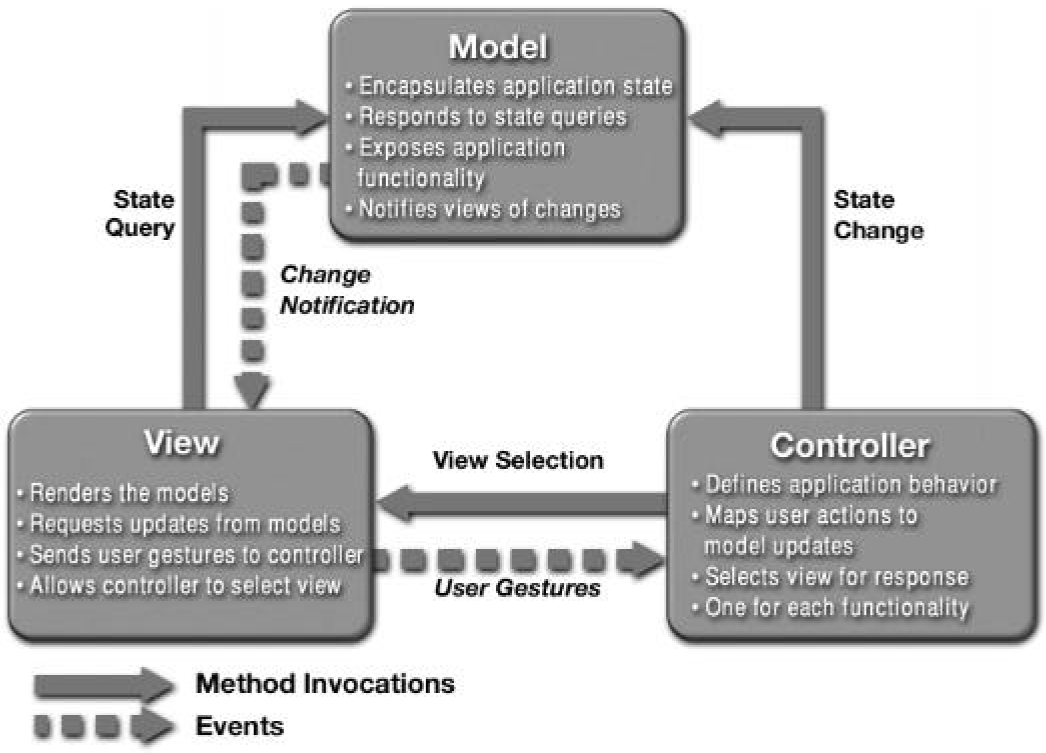
Model-View-Controller (MVC) J2EE architecture design pattern should be use for Web Application development. A MVC architecture is effective because of it allows different development groups to independently develop, test, deploy and modify the various components within an application. The MVC architecture can be summarized by the following three definitions:

- Model – The “Model”of an application is a set of objects that represents the business logic of the application. This usually includes classes to represent business abstractions (like Accounts, Purchases, etc.) as well as real-world objects (like Employees and Customers).

- View – A “View” is one particular way of presenting a set of information to a user. The best way to think of a “View” is to think of a particular web-page or “screen” that displays a single set of linked data to the user.

- Controller – The “Controller” layer in an application represents the parts of your application that handles the details of application flow and navigation. It translates information from the model layer into a form the View layer can understand, and deals with the all-important decisions as to what View to display next in response to a particular user action.

The duties these component types perform, and the relationships between components, appear below:


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

The key here is that the model is kept separate from the details of how the application is structured (the Controller) and how the information is presented to the user (the View). This allows JSP developers to change a view but not run the risk of corrupting the Controller code.

The MVC architecture has several revisions or standards defined.

- MVC model 1 basically used JSP pages for all display and control of an application

- MVC model 1.5 introduced JavaBeans to the JSP pages. Most control and logic is done in the JavaBean while JSPs still maintain the view.

- MVC model 2.0 introduces the concept of Servlets being the application controller component. JavaBeans (EJB's also) are used for the business logic of the application and JSP's are used for the view.

The MVC architecture will:

- Increase reusability by partially decoupling data presentation, data representation, and application operations.
- Enable multiple simultaneous data views.

- Facilitate maintenance, extensibility, flexibility, and encapsulation by decoupling software layers from one another.

The MVC model 2.0 is the architecture that should be used by SFA application teams. Several Frameworks exist that assist an application group in developing an MVC architecture. One of the more common frameworks is put out by the Apache Jakarta project and is called Struts (http://jakarta.apache.org/struts). This framework assists applications in dealing with MVC by helping to solve some common servlet issues that exist in most Web Application projects. These include:

- Mapping HTTP Parameters to JavaBeans – one of the most common tasks that servlet programmers have to do is maping a set of HTTP parameters (coming from the command line or from the POST of an HTML form) to a JavaBean for manipulation. This can be done using the <jsp:usebean> and <jsp:setProperty> tags, but this arrangement is cumbersome as it requires POSTing to a JSP, something that is not encouraged in a Model-II (MVC) architecture.

- Validation – There is no standard way in Servlet/JSP programming to validate that an HTML form is filled in correctly. This leaves every servlet programmer to develop his own validation procedures, or not, as is far too often the case.

- Error display – There is no standard way to handle the display of error messages in a JSP page or the generation of error messages in a servlet.

- Message Internationalization – Even when developers strive to keep as much of the HTML as possible in JSP's, there are often "hidden" obstacles to internationalization spread throughout Servlet and model code in the form of short error or informative text messages. While it is possible to introduce internationalization with the use of Java ResourceManagers, this is rarely done due to the complexity of adding these references.

- Hard coded JSP URI's – one of the more insidious problems in a servlet architecture is that the URI's of the JSP pages are usually coded directly into the code of the calling Servlet in the form of a static string reference used in the ServletContext.getRequestDispatcher() method. This means that it is impossible to reorganize the jsp's in a web site (or even change their names) without updating Java code in the servlets.

The problem is that programmers are too often faced with "reinventing the wheel" each time they begin building a new web-based application. Having a framework to do this kind of work for them would make developers more productive and allow them to focus more on the "essence" of the business problems they are trying to solve, rather than on the "accidents" of programming caused by the limitations of the technology.

9 Reference

Harvey W. Gunther, “WebSphere Application Server Development Best Practices for Performance and Scalability”, IBM WebSphere Whitepaper.

http://jakarta.apache.org/struts

http://java.sun.com/products/servlet/

http://java.sun.com/j2ee/blueprints/design_patterns/model_view_controller/

http://www-4.ibm.com/software/webservers/appserv/library.html

http://www.redbooks.ibm.com

JProbe Profiler With Memory Debugger, Developer’s Guide v3.0, Sitraka.

Joaquin Picon, et al. “Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server”, IBM RedBooks.